CLAIMS

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is as follows:

1. A method for recording and replaying execution of distributed programs on a computer system in a distributed environment, comprising:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

generating, for each execution thread, a logical thread schedule that identifies a sequence of said groups so as to allow deterministically replaying a non-deterministic arrival of stream socket connection requests, a non-deterministic number of bytes received during message reads, a non-deterministic binding of stream sockets to local ports, and a non-deterministic arrival of datagram messages.

2. The method according to claim 1, wherein a virtual machine in said distributed environment is modified to record events.

3. The method according to claim 2, wherein virtual machines in said distributed environment communicate with one another and events are recorded on each virtual machine.

4. The method according to claim 1, further comprising:

recording an arrival order of a message to guarantee the order and replay of applications.

5. The method according to claim 1, wherein said deterministically replaying includes:

modifying an implementation of a virtual machine of said distributed environment to record information on what transactions are occurring at an application level and using said information to replicate a same behavior in a replay.

6. The method according to claim 5, wherein an implementation of said virtual machine of said distributed environment is modified without changing an application being run.

7. The method according to claim 1, wherein said deterministically replaying includes:

recording events of a plurality of virtual machines having applications on each machine and having threads of a same application program running, said recording of said events providing a deterministic replay of events.

8. The method according to claim 1, wherein each of a plurality of virtual machines in said distributed environment records events at each said machine, and each said virtual machine communicates with one another, to guarantee the same execution order for the replay of any shared applications on said each virtual machine.

9. The method according to claim 1, wherein said application includes critical and non-critical events, and wherein said method further includes:

recording said critical events and logging a value of a global counter and a local counter, a single said global counter residing on a virtual machine, and said local counter residing on each thread of a virtual machine associated with said critical event.

10. The method according to claim 1, wherein said replay is based on logical thread schedules and logical intervals.

11. The method according to claim 1, wherein said replay is for a non-deterministic arrival of point-to-point stream socket connection requests.

12. The method according to claim 1, wherein said replay is for a non-deterministic number of bytes received during point-to-point stream socket message reads.

13. The method according to claim 1, wherein said replay is for a non-deterministic binding of stream sockets to local ports.

14. The method according to claim 1, wherein said replay is for a point-to-point datagram User Datagram Protocol (UDP) message sent to a single receiver.

15. The method according to claim 1, wherein said replay is for a point-to-points datagram User Datagram Protocol (UDP) message sent to multiple receivers.

16. The method according to claim 1, wherein said replay is for a non-deterministic arrival of number of bytes of point-to-point stream socket data.

17. The method according to claim 5, wherein said modified virtual machine is operable in a record mode such that logical thread schedule information and network interaction information of the execution are recorded while an application program runs, and in a replay mode, such that the execution behavior of the program is reproduced by enforcing the recorded logical thread schedule and the network interaction.

18. The method according to claim 1, wherein replaying of a multithreaded program includes:

capturing a logical thread schedule information during one execution of the program; and

enforcing an exact same schedule when replaying the execution.

19. The method according to claim 18, wherein said logical thread schedule comprises all of a plurality of physical thread schedules in an equivalence class.

20. The method according to claim 1, wherein each of said critical events represent one of a shared-variable access and a synchronization operation, said critical events affecting logical thread schedules,

said synchronization operation comprising one of a synchronized block and wait synchronization operation, a **monitorenter** synchronization operation, and a **monitorexit** synchronization operation,

wherein a different thread enters the critical section only after a first thread has executed the **monitorexit** operation.

21. The method according to claim 20, wherein said critical events further include **wait**, **notify**, and **notifyAll** synchronization operations for coordinating the execution order of multiple threads, and an **interrupt** synchronization operation that interrupts the execution of a thread at any point.

22. The method according to claim 1, wherein critical events comprise events whose execution order affect the execution behavior of the application,

wherein a logical thread schedule comprises a sequence of intervals of critical events, and wherein each interval corresponds to the critical and non-critical events executing consecutively in a specific thread.

23. The method according to claim 1, wherein for each given group of critical events, said critical events of the interval are consecutive, and only non-critical events can occur between consecutive critical events in the interval, and wherein said groups are ordered and no two adjacent intervals belong to the same thread.

24. The method according to claim 1, wherein only a critical event interval comprising a first critical event and a last critical event is traced and recorded.

25. The method according to claim 18, wherein critical events belonging to a given group are represented by an ordered pair of < *FirstCriticalEvent[i]* , *LastCriticalEvent[i]* >,

wherein *FirstCriticalEvent[i]* identifies the first critical event in the interval *i* and *LastCriticalEvent[i]* identifies the last critical event in the interval *i*.

26. The method according to claim 1, wherein the logical schedule interval *LSI[i]* corresponding to an interval *i* when the specific thread is scheduled for execution identifies the critical events that occur in the interval *i*.

27. The method according to claim 24, wherein a value of *FirstCriticalEvent[i]* and *LastCriticalEvent[i]* represent a global clock value that indicates the time that a corresponding event was executed, and is recorded,

wherein such global clock values identify the ordering of events in an execution stream..

28. The method according to claim 1, wherein each said critical event is identified by a global counter value that reflects an execution order of said critical events.

29. The method according to claim 1, wherein capturing logical thread schedule information is based on a global counter shared by all the threads and one local counter exclusively accessed by each thread,

wherein the global counter increments at each execution of a critical event to uniquely identify each critical event, and wherein a *FirstCEventi* and a *LastCEventi* are represented by their corresponding global counter values,

wherein the global counter is global within a particular virtual machine and each said virtual machine includes a different global counter, and a local counter increments at each execution of a critical event, such that a difference between the global counter and a thread's local counter is used to identify dynamically the logical schedule interval.

30. The method according to claim 1, wherein each critical event is uniquely associated with a global counter value, and wherein global counter values determine the order of critical events.

31. The method according to claim 30, wherein updating the global counter for a critical event and executing the critical event, are performed in one atomic operation for shared-variable accesses.

32. The method according to claim 1, wherein updating the global counter and executing the event both in one single atomic operation is only performed during the record phase.

33. The method according to claim 1, wherein for a thread to execute a schedule interval $LSI\ i$ = *<FirstCEventi ; LastCEventi>*, during the replay phase, the thread waits until the global counter value becomes the same as *FirstCEventi* without executing any critical events, and

when the global counter value equals *FirstCEventi*, the thread executes each critical event and also increments the global counter value until the value becomes the same as *LastCEventi*,

wherein when the global counter value equals *LastCEventi*, the thread fetches its next schedule interval, *LSI i+1 = <FirstCEventi+1 ; LastCEventi+1 >*, from the log and waits until the global counter value becomes the same as *FirstCEventi+1*, an operation being repeated until no more schedule intervals exist in the log.

34. The method according to claim 1, wherein for point-to-point communication, a socket is created, and *getInputStream()* and *getOutputStream()* of the Socket object return InputStream and OutputStream objects to be used for reading via *a read()* method call and writing via *a write()* method call stream data over the socket stream,

wherein a plurality of socket application programming interfaces (APIs) are provided including socket APIs for listening for connections on a stream socket via a *listen()* method call, binding a socket to a local port via a *bind()* method call, and determining the number of bytes that can be read without blocking via an *available()* method call,

wherein each stream socket call including *accept(), bind, create(), listen(), connect(), close(), available(), read(),* and *write()* is mapped into a native method call in a virtual machine implementation.

35. The method according to claim 1, wherein deterministic replay of network events comprises deterministic re-establishment of socket connections among threads.

36. The method according to claim 1, wherein each virtual machine is assigned a unique virtual machine identity (VM-id) during a record phase,

said identity being logged in the record phase and reused in a replay phase, to allow identification of a sender of a message or connection request.

37. The method according to claim 1, wherein critical events on a virtual machine are identified by their global counter value on the virtual machine, and a *networkEventId* is used to uniquely identify a network event in a distributed application,

wherein said *networkEventId* is defined as a tuple <*threadNum, eventNum*>, where *threadNum* is a thread number of a specific thread executing the network event and *eventNum* is a number that identifies the network event within the thread, said *eventNum* being used to order network events within a specific thread.

38. The method according to claim 37,wherein a *connectionId* is for identifying a connection request at a connect network event,

said *connectionId* is a tuple,<*dJVMId, threadNum*>, where *dJVMId* is the identity of the virtual machine at which the connect event is being generated, and *threadNum* is the thread number of the client thread generating the connection request.

39. The method accoridng to claim 38, wherein said *threadNum* has a same value in both the record and replay phases, and wherein events are sequentially ordered within a thread, and the

eventNum of a particular network event executed by a particular thread is guaranteed to be the same in the record and replay phases.

40. The method according to claim 1, wherein network operations are marked as critical events, thereby allowing threads performing operations on different sockets to proceed in parallel.

41. The method according to claim 39, wherein, in the record phase, at the connect, a virtual machine-client sends a socket-connection request to a server,

when the socket connection is established, the client thread on the virtual machine-client sends the *connectionId* for the connect over the established socket as the first data,

42. The method according to claim 39, wherein, in the replay phase, the virtual machine-client executes the *connect* and sends the *connectionId* of the *connect* to the server as the first data, said *connect* being a critical event, such that the virtual machine-client ensures that the connect call returns only when the *globalCounter* for this critical event is reached.

43. The method according to claim 42, wherein, on the server side, during the record phase, at an accept, the virtual machine-server accepts the connection and receives the *connectionId* sent by the client as the first data at the corresponding connect, the virtual machine-server logging information about the connection established into the *NetworkLogFile,*

wherein for each successful accept call, the log contains a *ServerSocketEntry*, said *ServerSocketEntry* comprising a tuple, *<serverId , clientId >*, wherein said *serverId* is the

networkEventId of the corresponding *accept* event and wherein said *clientId* is the *connectionId* sent by the virtual machine-client.

44. The method according to claim 1, wherein during a record phase having a client that performs a connect and a server that performs an accept, a method comprising:

on the client side, performing a recording of critical events including *enterGCCriticalSection*, updating *ClientGC*, and *leaveGCCriticalSection*;

sending a connection request to the server side;

on the client side, sending the *ClientEventID* as a tuple comprising <*clientJVMId, ClientGC*> to the server;

on the server side, receiving the *ClientEventID* and logging, by the server side, <*ServerGC, ClientEventID*> into a *ServerSocketLog*.

45. The method according to claim 44, wherein, for replaying accept events, a virtual machine includes a connection pool for buffering out-of-order connections,

wherein during the replay phase, when an *accept* is executed by a server thread *t_s* on the virtual machine-server, said virtual machine-server identifies the *networkEventId* for the accept event, and

wherein the *connectionId* is identified from the *NetworkLogFile* with matching *networkEventId* value, and said virtual machine-server checks the connection pool to determine whether a Socket object has been created with the matching *connectionId*.

46. The method accoridng to claim 45, wherein if the Socket object has been created, the Socket object is returned by said virtual machine-server to complete the accept, and

wherein if the Socket object has not already been created with the matching *connectionId*, the virtual machine-server buffers information about out-of-order connections in the connection pool until said virtual machine-server receives a connection request with the matching *connectionId*, said virtual machine-server then creating and returning a Socket object for the matching connection.

47. The method according to claim 45, wherein an accept on the server side during the replay mode includes:

recording critical events;

retrieving a *ClientEventID* which equals *recValue* from the *ServerSocketLog* for a respective *ServerGC*;

checking the connection pool for the *recValue*, wherein if the *recValue* is in the connection pool, then the process ends, and

wherein if the *recValue* is not in the connection pool, then the method further comprises:

accepting a connection request and receiving the *ClientEventId* by the server; and

determining whether the *ClientEventId* is not equal to the *recValue*, and if not equal, then saving the *ClientEventId* in the connection pool, and if it is determined that the ClientEventId is equal to the *recValue*, then the process ends.

48. The method according to claim 47, wherein socket read and write events are identified as critical events in a virtual machine, and the virtual machine's global counter is updated for each of these calls,

wherein in the record phase, the virtual machine executes the read and logs the thread-specific *eventNum* and number of bytes read *numRecorded* in the *NetworkLogFile*.

49. The method according to claim 48, wherein in the replay phase, at a corresponding read event, the virtual machine reads only the *num-Recorded* number of bytes even if more bytes are available to read.

50. The method according to claim 48, wherein in the replay phase, at the corresponding read event, the virtual machine thread retrieves the *numRecorded* number of bytes from the *NetworkLogFile* corresponding to the current *eventNum* and the thread reads only the *numRecorded* bytes even if more bytes are available to read, or will block until *numRecorded* bytes are available to read, and

wherein the execution returns from the read call only when the *globalCounter* for the critical event is reached.

51. The method according to claim 50, wherein during a record mode, for a *read()* method call, the *read* event is executed, returning "n"bytes read which is logged in a recordedValue, and the critical event corresponding to the read is logged.

52. The method according to claim 50, wherein during a replay mode for a *read()* method call, the *read* critical event is executed returning the number n of bytes read,

wherein if n is greater than the recorded value, the *read* critical event is executed again, and wherein if n is less than the recorded value, the *read* critical event is executed again and bytes are read until the *recordedValue* number of bytes are read,

wherein when n is equal to *recordedValue*, then the *read* critical event is recorded by performing a *enterGcCriticalSection* and *leaveGcCriticalSection*, and the process stops.

53. The method according to claim 53, wherein for multiple *writes* on a same socket, replaying of the *writes* to the same socket occur in a same order and all *reads* from the socket read the bytes in a same order in both record and replay modes.

54. The method according to claim 53, wherein an occurrence of multiple writes to a same socket are recorded, without recording other events that do not operate on the same socket,

wherein said events that do use the same socket are blocked by using a lock variable for each socket.

55. The method according to claim 54, wherein enterFDCriticalSection(socket) allows only *reads* or *writes* corresponding to that socket to execute the code therein, thereby preserves an execution ordering of different critical events.

56. The method according to claim 53, wherein *available* and *bind* call events comprise critical events, and implement a network query,

wherein said *available* call checks a number of bytes available on the stream socket, and said *bind* call returns a local port to which the socket is bound,

wherein said *available* call comprises a blocking call, and in the record phase, is executed before enter a GC-critical section, and the virtual machine records a number of bytes available,

wherein in the replay phase, the *available* call event blocks until it returns the recorded number of bytes available on the stream socket,

wherein said *bind* event, in the record phase, is executed within the GC-critical section and the virtual machine records its return value.

57. The method according to claim 1, wherein user datagram protocol sockets are created via a *DatagramSocket* class, and

wherein during a record phase, a sender virtual machine intercepts a datagram sent by an application and inserts a `DGNETEventId` of a send event at an end of a data segment of the application datagram, and the virtual machine increases a length field of the datagram to include an added size for a datagram identification.

58. The method according to claim 57, wherein if the datagram is larger than a predetermined size, said datagram is split, with each split datagrams carrying the same `DGNETEventId`, and different tags including one of `FRONT_UDP` and `REAR_UDP`, to indicate one of a front portion or rear portion.

59. The method according to claim 58, wherein if said datagram is less than or equal to the predetermined size, then the datagram carries information that it is a whole datagram.

60. A method for supporting execution replay with respect to a stream socket Application programming interface (API) comprising:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

deterministically replaying non-deterministic arrival of stream socket connection requests, non-deterministic number of bytes received during message reads, and non-deterministic binding of stream sockets to local ports.

61. A method for supporting execution replay with respect to datagram socket Application Programming Interface (API) including:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread;

determining out-of-order delivery of packets; and

determining a non-deterministic number of packets delivered during different executions of the same program.

62. The method according to claim 61, wherein packets are sent to multiple receivers.

63. The method according to claim 62, wherein said replaying allows repeating the exact behavior of thread execution and events in a distributed environment.

64. A software facility for allowing recording and replaying execution of distributed programs on a computer system in a distributed environment, comprising:

a first module for identifying an execution order of critical events of a program;

a second module for generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

a third module for generating, for each execution thread, a logical thread schedule that identifies a sequence of said groups so as to allow deterministically replaying a non-deterministic arrival of stream socket connection requests, a non-deterministic number of bytes received during message reads, a non-deterministic binding of stream sockets to local ports, and a non-deterministic arrival of datagram messages.

65. A software facility for supporting execution replay with respect to a stream socket Application programming interface (API) comprising:

a first module for identifying an execution order of critical events of a program;

a second module for generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

a third module for deterministically replaying non-deterministic arrival of stream socket connection requests, non-deterministic number of bytes received during message reads, and non-deterministic binding of stream sockets to local ports.

66. A software facility for supporting execution replay with respect to datagram socket Application Programming Interface (API) including:

a first module for identifying an execution order of critical events of a program;

a second module for generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread;

a third module for determining out-of-order delivery of packets; and

a fourth module for determining a non-deterministic number of packets delivered during different executions of the same program.

67. A programmable storage medium tangibly embodying a program of machine-readable instructions executable by a digital processing apparatus to perform a method of recording and replaying execution of distributed programs on a computer system in a distributed environment, said method comprising:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

generating, for each execution thread, a logical thread schedule that identifies a sequence of said groups so as to allow deterministically replaying a non-deterministic arrival of stream

socket connection requests, a non-deterministic number of bytes received during message reads, a non-deterministic binding of stream sockets to local ports, and a non-deterministic arrival of datagram messages.

68. A programmable storage medium tangibly embodying a program of machine-readable instructions executable by a digital processing apparatus to perform a method for supporting execution replay with respect to a stream socket Application programming interface (API), said method comprising:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread; and

deterministically replaying non-deterministic arrival of stream socket connection requests, non-deterministic number of bytes received during message reads, and non-deterministic binding of stream sockets to local ports.

69. A programmable storage medium tangibly embodying a program of machine-readable instructions executable by a digital processing apparatus to perform a method for supporting execution replay with respect to datagram socket Application Programming Interface (API), said method including:

identifying an execution order of critical events of a program;

generating groups of critical events of said program, wherein for each group, critical events belonging to said group belong to a common execution thread;

determining out-of-order delivery of packets; and

determining a non-deterministic number of packets delivered during different executions of the same program.